Binary Codes: From Symbols to Binary Codes*

Louis Scharf

NOTE: This module is part of the collection, *A First Course in Electrical and Computer Engineering*. The LaTeX source files for this collection were created using an optical character recognition technology, and because of this process there may be more errors than usual. Please contact us if you discover any errors.

Perhaps the most fundamental idea in communication theory is that arbitrary symbols may be represented by strings of binary digits. These strings are called binary words, binary addresses, or binary codes. In the simplest of cases, a finite alphabet consisting of the letters or symbols $s_0, s_1, ..., s_{M-1}$ is represented by binary codes. The obvious way to implement the representation is to let the i^{th} binary code be the binary representation for the subscript i:

$$\begin{aligned} s_0 \sim 000 &= a_0 \\ s_1 \sim 001 &= a_1 \\ &\vdots \\ s_6 \sim 110 &= a_6 \\ s_7 \sim 111 &= a_7. \end{aligned} \tag{1}$$

The number of bits required for the binary code is N where

$$2^{N-1} < M \leq 2^N. \tag{2}$$

We say, roughly, that $N = log_2 M$.

Octal Codes. When the number of symbols is large and the corresponding binary codes contain many bits, then we typically group the bits into groups of three and replace the binary code by its corresponding

*Version 1.7: Sep 16, 2009 12:59 pm GMT-5
†http://creativecommons.org/licenses/by/3.0/

octal code. For example, a seven-bit binary code maps into a three-digit octal code as follows:

$$\begin{aligned} 0000000 &\sim 000 \\ 0000001 &\sim 001 \\ &\vdots \\ 0100110 &\sim 046 \\ &\vdots \\ 101111 &\sim 137 \\ &\vdots \\ 1111111 &\sim 177. \end{aligned} \tag{3}$$

The octal ASCII codes for representing letters, numbers, and special characters are tabulated in Table 1 (Table 1).

Exercise 1

Write out the seven-bit ASCII codes for $A, q, 7, and \{$.

	'0	*'1*	*'2*	*'3*	*'4*	*'5*	*'6*	*'7*
'00x	[U+2400]	[U+2401]	[U+2402]	[U+2403]	[U+2404]	[U+2405]	[U+2406]	[U+2407]
'01x	[U+2408]	[U+2409]	[U+240A]	[U+240B]	[U+240C]	[U+240D]	[U+240E]	[U+240F]
'02x	[U+2410]	[U+2411]	[U+2412]	[U+2413]	[U+2414]	[U+2415]	[U+2416]	[U+2417]
'03x	[U+2418]	[U+2419]	[U+241A]	[U+241B]	[U+241C]	[U+241D]	[U+241E]	[U+241F]
'04x	[U+2420]	!	"	#	$	%	&	’
'05x	(	)	*	+	,	-	.	/
'06x	0	1	2	3	4	5	6	7
'07x	8	9	:	;	<	=	>	?
'10x	@	A	B	C	D	E	F	G
'11x	H	I	J	K	L	M	N	O
'12x	P	Q	R	S	T	U	V	W
'13x	X	Y	Z	[	\	]	^	_
'14x	‘	a	b	c	d	e	f	g
'15x	h	i	j	k	l	m	n	o
'16x	p	q	r	s	t	u	v	w
'17x	x	y	z	{	|	}	~	[U+2421]

Table 1: Octal ASCII Codes (from Donald E. Knuth, *The TEXbook*, ©1986 by the American Mathematical Society, Providence, Rhode Island p. 367, published by Addison-Wesley Publishing Co.)

Exercise 2

Add a 1 or a 0 to the most significant (left-most) position of the seven-bit ASCII code to produce an eight-bit code that has even parity (even number of 1's). Give the resulting eight-bit ASCII codes and the corresponding three-digit octal codes for %, $u, f, 8$, and +.

Quantizers and A/D Converters. What if the source alphabet is infinite? Our only hope is to approximate it with a finite collection of finite binary words. For example, suppose the output of the source is an analog voltage that lies between $-V_0$ and $+V_0$. We might break this peak-to-peak range up into little voltage cells of size $\frac{2V}{M}A$ and approximate the voltage in each cell by its midpoint. This scheme is illustrated in Figure 1 (Figure 1). In the figure, the cell C_i is defined to be the set of voltages that fall between $i\frac{2V}{M-}pV_{\Delta}$ and $i^{2V}M\Delta + \frac{V}{-}M^{A}$:

$$C_i = \{V : i.\frac{2V_0}{M} - \frac{V_0}{M} < V \leq i\frac{2V_0}{M} + \frac{V_0}{M}\}. \tag{4}$$

The mapping from continuous values of V to a finite set of approximations is

$$Q(V) = i\frac{2V_0}{M}, \text{ if } V \in C_i. \tag{5}$$

That is, V is replaced by the quantized approximation $i\frac{2V_0}{M}$ whenever V lies in cell C_i. We may represent the quantized values $i\frac{2V}{M}0$ with binary codes by simply representing the subscript of the cell by a binary word. In a subsequent course on digital electronics and microprocessors you will study A/D (analog-to-digital) converters for quantizing variables.

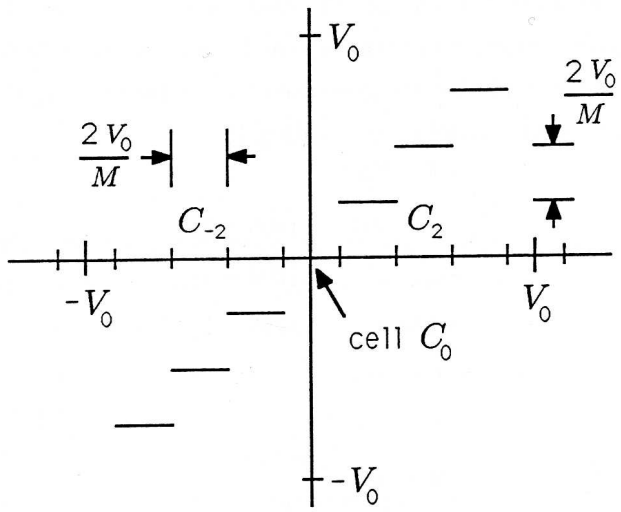


Figure 1: A Quantizer

Example 1
If $M = 8$, corresponding to a three-bit quantizer, we may associate quantizer cells and quantized

levels with binary codes as follows:

$$\begin{aligned} V \in C_{-3} &\Rightarrow V_{-3} = (-3)\,\tfrac{2V_0}{8} \sim 111 \\ V \in C_{-2} &\Rightarrow V_{-2} = (-2)\,\tfrac{2V_0}{8} \sim 110 \\ V \in C_{-1} &\Rightarrow V_{-1} = (-1)\,\tfrac{2V_0}{8} \sim 101 \\ V \in C_0 &\Rightarrow V_0 = 0 \sim 000 \\ V \in C_1 &\Rightarrow V_1 = (1)\,\tfrac{2V_0}{8} \sim 001 \\ V \in C_2 &\Rightarrow V_2 = (2)\,\tfrac{2V_0}{8} \sim 010 \\ V \in C_3 &\Rightarrow V_3 = (3)\,\tfrac{2V_0}{8} \sim 011. \end{aligned} \tag{6}$$

This particular code is called a *sign-magnitude code*, wherein the most significant bit is a sign bit and the remaining bits are magnitude bits (e.g., $110 \sim -2$ and $010 \sim 2$). One of the defects of the sign-magnitude code is that it wastes one code by using 000 for 0 and 100 for-O. An alternative code that has many other advantages is the 2's *complement code*. The $2^{'}s$ complement codes for positive numbers are the same as the sign-magnitude codes, but the codes for negative numbers are generated by complementing all bits for the corresponding positive number and adding 1:

$$\begin{aligned} -4 &\sim 100 \\ -3 &\sim 101 \quad (100+1) \\ -2 &\sim 110 \quad (101+1) \\ -1 &\sim 111 \quad (110+1) \\ 0 &\sim 000 \\ 1 &\sim 001 \\ 2 &\sim 010 \\ 3 &\sim 011. \end{aligned} \tag{7}$$

Exercise 3
Generate the four-bit sign-magnitude and four-bit 2's complement binary codes for the numbers $-8, -7, ..., -1, 0, 1, 2, ..., 7$.

Exercise 4
Prove that, in the $2^{'}s$ complement representation, the binary codes for $-nand + n$ sum to zero. For example,

$$\begin{aligned} &101 + 011 = 000 \\ &(-3) \quad (3) \quad (0)\,. \end{aligned} \tag{8}$$

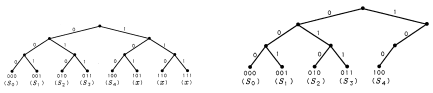
In your courses on computer arithmetic you will learn how to do arithmetic in various binary-coded systems. The following problem illustrates how easy arithmetic is in 2's complement.

Exercise 5
Generate a table of sums for all $2^{'}s$ complement numbers between -4 and $+3$. Show that the sums are correct. Use $0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1$, and $1 + 1 = 0$ with a carry into the next bit. For example, $001 + 001 = 010$.

Binary Trees and Variable-Length Codes. The codes we have constructed so far are constant-length codes for finite alphabets that contain exactly $M = 2^N$ symbols. In the case where $M = 8$ and $N = 3$, then

the eight possible three-bit codes may be represented as leaves on the branching tree illustrated in Figure 2(a) (Figure 2). The tree grows a left branch for a 0 and a right branch for a 1, until it terminates after three branchings. The three-bit codes we have studied so far reside at the terminating leaves of the binary tree. But what if our source alphabet contains just five symbols or letters? We can represent these five symbols as the three-bit symbols 000 through 100 on the binary tree. This generates a constant-length code with three unused, or illegal, symbols 101 through 111. These are marked with an "x" in Figure 2(a) (Figure 2). These unused leaves and the branches leading to them may be pruned to produce the binary tree of Figure 2(b) (Figure 2).

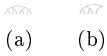
If we admit variable-length codes, then we have several other options for using a binary tree to construct binary codes. Two of these codes and their corresponding binary trees are illustrated in Figure 3 (Figure 3). If we disabuse ourselves of the notion that each code word must contain three or fewer bits, then we may construct binary trees like those of Figure 4 (Figure 4) and generate their corresponding binary codes. In Figure 4(a) (Figure 4), we grow a right branch after each left branch and label each leaf with a code word. In Figure 4(b) (Figure 4), we prune off the last right branch and associatea code word with the leaf on the last left branch.



(a) (b)

Figure 2: Binary Trees and Constant-Length Codes; (a) Binary Tree, and (b) Pruned Binary Tree

All of the codes we have generated so far are organized in Table 2 (Table 2). For each code, the average number of bits/symbol is tabulated. This average ranges from 2.4 to 3.0. If all symbols are equally likely to appear, then the best variable-length code would be code 2.



(a) (b)

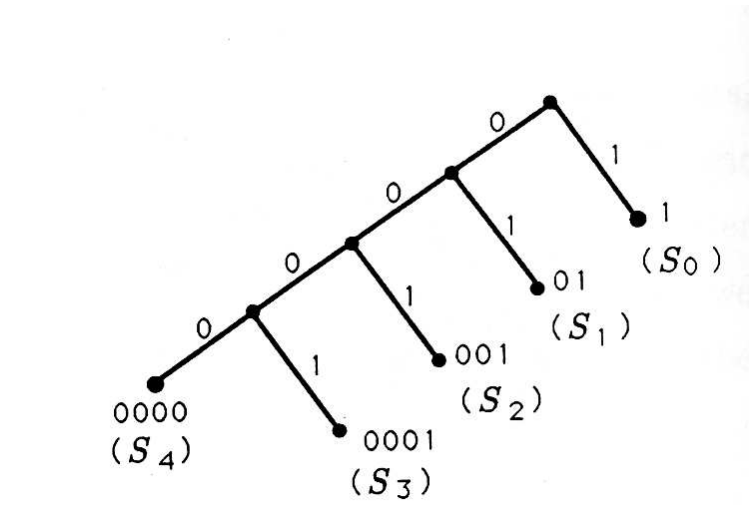
Figure 3: Binary Trees and Variable-Length Codes; (a) Binary Tree for Variable-length Code, and (b) Another Binary Tree for Variable-length Code

All of the codes we have constructed have a common characteristic: each code word is a terminating leaf on a binary tree, meaning that no code word lies along a limb of branches to another code word. We say that no code word is a *prefix* to another code word. This property makes each of the codes *instantaneously decodable*, meaning that each bit in a string of bits may be processed instantaneously (or independently) without dependence on subsequent bits.

Exercise 6

Decode the following sequence of bits using code 2:

$$01110011110000001011001111. \tag{9}$$



(a) (b)

Figure 4: Left-Handed Binary Trees for Variable-Length Codes; (a) Left-handed Binary Tree, and (b) Pruned Binary Tree

Code #	S_0	S_1	S_2	S_3	S_4	**Average Bits/Symbol**
1	000	001	010	011	100	$15/5 = 3.0$
2	000	001	01	10	11	$12/5 = 2.4$
3	000	001	010	011	1	$13/5 = 2.6$
4	1	01	001	0001	00001	$15/5 = 3.0$
5	1	01	001	0001	0000	$14/5 = 2.8$

Table 2: Variable Length Codes

Exercise 7

Illustrate the following codes on a binary tree. Which of them are instantaneously decodable? Which can be pruned and remain instantaneously decodable?

$$\begin{array}{lllll} S_0 & S_1 & S_2 & S_3 & S_4 \\ 011 & 100 & 00 & 11 & 101 \\ 011 & 100 & 00 & 0 & 01 \\ 010 & 000 & 100 & 101 & 111. \end{array} \tag{10}$$

Code #2 generated in Table 2 (Table 2) seems like a better code than code #5 because its average number of bits/symbol (2.4) is smaller. But what if symbol S_0 is a very likely symbol and symbol S_4 is a very unlikely

one? Then it may well turn out that the average number of bits used by code #5 is less than the average number used by code #2. So what is the best code? The answer depends on the relative frequency of use for each symbol. We explore this question in the next section.